**Lamda Expression**

* In all programs, there are two primary components: the data (the stuff a program knows) and the behaviours (the stuff a program can do to/with that data). OOP says that bringing together data and its associated behaviour in a single location (called an “object”) makes it easier to understand how a program works. FP says that data and behaviour are distinctively different things and should be kept separate for clarity.
* If we want a block of code to be executed, we need to create an object and pass the object around, such as using [the strategy design pattern](https://www.programcreek.com/2011/01/a-java-example-of-strategy-design-pattern/). From Java 8, lambda expressions enable us to treat functionality as method argument and pass a block of code around. Lambda expressions in Java 8 are very powerful and therefore very compelling.
* Lambda expressions basically express instances of [functional interfaces](https://www.geeksforgeeks.org/functional-interfaces-java/) (An interface with single abstract method is called functional interface. An example is java.lang.Runnable). lambda expressions implement the only abstract function and therefore implement functional interfaces
* In programming, a Lambda expression (or function) is just an anonymous function, i.e., a function with no name and without being bounded to an identifier.
* lambda expressions are nameless functions given as constant values, and written exactly in the place where it’s needed, typically as a parameter to some other function.

**Functional interface**

* In [java 8](https://howtodoinjava.com/category/java8/) which permit exactly one abstract method inside them. These interfaces are also called Single Abstract Method interfaces (SAM Interfaces). These can be represented using Lambda expressions, Method reference and constructor references as well. Some in built functional interfaces are Predicate, Consumer,Function etc.
* Function<T, R> is an in-built functional interface introduced in Java 8 in the java.util.function package. The primary purpose for which Function<T, R> has been created is for mapping scenarios i.e when an object of a type is taken as input and it is converted(or mapped) to another type.
* Function<T, R>’s Function Descriptor is T -> R. This means an object of type T is input to the lambda and an object of type R is obtained as return value.

**Java Stream**

* Introduced in Java 8, the Stream API is used to process collections of objects. A stream is a sequence of objects that supports various methods which can be pipelined to produce the desired result.
  + A stream is not a data structure instead it takes input from the Collections, Arrays or I/O channels.
  + Streams don’t change the original data structure, they only provide the result as per the pipelined methods.
  + Each intermediate operation is lazily executed and returns a stream as a result, hence various intermediate operations can be pipelined. Terminal operations mark the end of the stream and return the result.

**Predicates**

* In Java 8, [Predicate](https://docs.oracle.com/javase/8/docs/api/java/util/function/Predicate.html) is a [functional interface](https://howtodoinjava.com/java8/functional-interface-tutorial/) and can therefore be used as the assignment target for a [lambda expression](https://howtodoinjava.com/java8/complete-lambda-expressions-tutorial-in-java/) or method reference.

**Method reference**

In [Java 8](https://howtodoinjava.com/java-8-tutorial/), we can refer a method from class or object using **class::methodName** type syntax.

**PermGen Error Removed**

By default, the initial size of the Metaspace will be set to 12 MB and 16 MB on the 32-bit client and server VMs, respectively, with larger sizes for the 64-bit VMs, and the maximum size is limited only by the amount of virtual memory available. These initial and maximum sizes can be capped manually, however, using the new MetaspaceSize and MaxMetaspaceSize flags, which are analogous to the old PermSize and MaxPermSize flags (note that the PermSize and MaxPermSize flags will be ignored at startup, and a warning will be issued).

 At runtime, the actual allocation of the Metaspace will be dynamically re-sized to meet the application’s needs. Once its use reaches MetaspaceSize, garbage collection will be triggered and the Metaspace will be expanded. This will continue to occur until either MaxMetaspaceSize is reached (if set) or if it runs out of native memory. At this point, a java.lang.OutOfMemoryError:Metaspace error will be thrown.

**Default Methods**

* Default methods enable you to add new functionality to the interfaces of your libraries and ensure binary compatibility with code written for older versions of those interfaces.
* Simplest answer is to enable the functionality of lambda expression in java. Lambda expressions are essentially of type of functional interface. To support lambda expressions seamlessly, all core classes have to be modified.
* The default methods were introduced to provide backward compatibility so that existing interfaces can use the lambda expressions without implementing the methods in the implementation class.

**Conflict resolution rules for inherited default methods**

* Any method inherited from a class or a superclass is invoked over any default method inherited from an interface.
* If default methods with the same method signature exist in an interface and its child interfaces, then the default method from the child interface is invoked
* In case Rule 1 and Rule 2 are not able to resolve the conflict then the implementing class has to specifically override and provide a method with the same method definition

**Difference between default method and static method in Interface**

* Default methods **can be** overridden in implementing class, while static **cannot**.
* Static method belongs **only** to Interface class, so you can only invoke static method on Interface class, not on class implementing this Interface.

MyClass.staticMethod(); //not valid - static method may be invoked on containing interface class only

MyInterface.staticMethod(); //valid

**Consumer**

* Consumer<T> is an in-built functional interface introduced in Java 8 in the java.util.function package. Consumer can be used in all contexts where an object needs to be consumed,i.e. taken as input, and some operation is to be performed on the object without returning any result.
* Common example of such an operation is *printing* where an object is taken as input to the printing function and the value of the object is printed

**ExecutorService**

* The Java Executor Service is a construct that allows you to pass a task to be executed by a thread asynchronously.
* The executor service creates and maintains a reusable pool of threads for executing submitted tasks.
* The service also manages a queue, which is used when there are more tasks than the number of threads in the pool and there is a need to queue up tasks until there is a free thread available to execute the task.
* The **Java Executor Framework** has been introduced in Java 1.5 and it is a part of java concurrency package. The Executor framework is an abstraction layer over the actual implementation of java multithreading.
* It is used for standardizing invocation, scheduling, execution and control of asynchronous tasks in parallel threads.
* **Executor implementation** in java uses thread pools which consists of worker threads. The entire management of worker threads is handled by the framework. So the overhead in memory management is much reduced compared to earlier multithreading approaches.
* The **Java Executor framework** creates tasks by using instances of Runnable or Callable. In case of Runnable, the run () method does not return a value or throw any checked exception.
* But Callable is a more functional version in that area. It defines a call () method that allows the return of some computed value which can be used in future processing and it also throws an exception if necessary.
* [Java Concurrency API](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/package-summary.html) defines the following three executor interfaces that covers everything that is needed for creating and managing threads -
* **Executor** - A simple interface that contains a method called execute() to launch a task specified by a Runnable object.
* **ExecutorService** - A sub-interface of Executor that adds functionality to manage the lifecycle of the tasks. It also provides a submit() method whose overloaded versions can accept a Runnable as well as a Callable object. Callable objects are similar to Runnable except that the task specified by a Callable object can also return a value. We’ll learn about Callable in more detail, in the [next blog post](https://www.callicoder.com/java-callable-and-future-tutorial/).
* **ScheduledExecutorService** - A sub-interface of ExecutorService. It adds functionality to schedule the execution of the tasks.

**Create one or more tasks and put in the queue**

* After creating the executor now it’s time for creating tasks. Create one or more tasks to be performed as instances of either Runnable or Callable. In this framework, all the tasks are created and populated in a queue. After the task creation is complete the populated queue is submitted for concurrent execution.

**Submit the task to the Executor**

* After creating the ExecutorService and proposed tasks, you need to submit the task to the executor by using either submit () or execute () method. Now as per your configuration the tasks will be picked up from the queue and run concurrently.
* Both submit() and execute() methods are used to submit a task to [Executor framework](http://javarevisited.blogspot.com/2013/07/how-to-create-thread-pools-in-java-executors-framework-example-tutorial.html) for asynchronous execution.
* **The submit()** can accept both [Runnable](http://java67.blogspot.com/2016/01/7-differences-between-extends-thread-vs-implements-Runnable-java.html) and [Callable](http://javarevisited.blogspot.com/2015/06/how-to-use-callable-and-future-in-java.html) task but **execute()** can only accept the Runnable task. The return type of **submit()** method is a Future object but return type of **execute()** method is void.
* scheduledExecutorService.schedule() function takes a Runnable, a delay value, and the unit of the delay.

**Execute the task**

* Next the actual execution of the tasks will be managed by the framework. The Executor is responsible for managing the task’s execution, thread pool, synchronization and queue.

**Shutdown the Executor**

* The termination is executed by invoking its shutdown () method. You can choose to terminate it gracefully, or abruptly.
* **shutdown()**- when shutdown() method is called on an executor service, it stops accepting new tasks, waits for previously submitted tasks to execute, and then terminates the executor.
* **shutdownNow()** - this method interrupts the running task and shuts down the executor immediately.

------------------------------------------------------------------------------------------------------------------------------

* Well, Java provides a Callable interface to define tasks that return a result. A Callable is similar to Runnable except that it can return a result and throw a checked exception

**Spliterator**

* **Java Spliterator** interface is an internal iterator that breaks the [stream](https://howtodoinjava.com/java8/java-8-tutorial-streams-by-examples/) into the smaller parts. These smaller parts can be processed in parallel.
* The Java collection classes provide *default* **stream()** and **parallelStream()** methods which internally use the Spliterator through the call to the spliterator(). It helps in processing the collection data in parallel.
* It provides support for parallel processing of stream of elements for any collection.

**ParallelStream**

* Parallel stream enables parallel computing that involves processing elements concurrently in parallel with each element in a seperate thread. But this does not guarantee high performance and faster execution everytime. It again depends on the number of CPU cores available.
* Parallelstream has a much higher overhead compared to a sequential one. Parallel stream should be used only when there is a huge amount of elements to process and there is a performance problem with processing these elements. Also the parallel stream operations should be independent.

**CompletableFuture**

* CompletableFuture is used for asynchronous programming in Java. Asynchronous programming is a means of writing non-blocking code by running a task on a separate thread than the main application thread and notifying the main thread about its progress, completion or failure.
* This way, your main thread does not block/wait for the completion of the task and it can execute other tasks in parallel.
* Having this kind of parallelism greatly improves the performance of your programs.